Convolution

Classify Clash of Clans Bases

Alexander Garcia

May 2025

Contents

1 Abstract **3**

2 Dataset **3**

3 Overfitting **4**

4 Build Basic CNNs **5**
4.1 GlobalAverage vs Flatten 5
4.2 Model Comparisons 6

5 Augmentation **7**
5.1 Examples of Augmenting 8
5.2 Effects of Augmenting 9

6 Regularization **10**
6.1 Effects of Regularization 10
6.2 Table of Validation Accuracy 11

7 Residual Networks **12**
7.1 Effects of Residual Connections 12
7.2 ResNet 13

8 Conclusion **14**

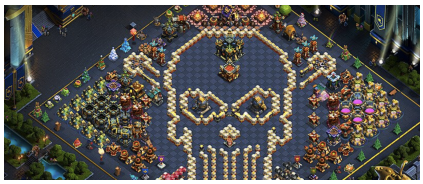
1 Abstract

The goal of this project was to research how to build a convolutional neural network (CNN). To build a proper network, image scraping and data analysis, overfitting, model architecture selection, effects of regularization and augmentation, and use of AI to build a model were all studied, and their impacts are noted throughout this report. During each step, the metric used to determine success is often validation accuracy with a final check against unseen test data. These tasks are all necessary to understand how a CNN operates and the impact each technique has on the final results. Some techniques proved to be useful, such as reducing learning rate on the optmizer and adjusting padding while others, augmentation and regularization, were left out of the final model due to no gains on validation accuracy and in some cases significant performance drop or overfitting occurred. The model that performed best after all tasks was a 5-layer convolution increasing filters as the network deepens with a total parameter count of 1.6 million.

2 Dataset

The dataset is images of Clash of Clans, a popular mobile strategy game that features an area of 44 x 44 tiles where players can fully customize their base. Players are free to place buildings anywhere in any configuration within this area and the amount of buildings available depends on the player's Town Hall level. There are 17 total Town Halls at the time of this report, but I'm only focused on classifying Town Hall 10 through Town Hall 17.

(a) Town Hall 10



(b) Town Hall 17

Figure 1: Different base layouts showing a war setup on the left and art on the right

Town Hall	Train	Validation	Test	Total
th_10	90	20	20	130
th_11	90	20	20	130
th_12	90	20	20	130
th_13	90	20	20	130
th_14	90	20	20	130
th_15	98	22	22	142
th_16	81	18	18	117
th_17	93	20	21	134

Table 1: Image Distribution

3 Overfitting

The goal of task 1 was to find the smallest architecture that could overfit the **entire** dataset. The dataset was not split into test, train, and validation, but was trained together. For this task, the Dense layer was not supposed to do most of the work, that is, the proportion of parameters in the Dense layer should not exceed 10-20% of the total parameters. I studied the effects of models with as low as 187k parameters up to 5.7 million parameters. I was unable to reach 100%, but had a few models over 90%.

Conv Layers	Dense Neurons	# of Parameters	Accuracy
3	10	187k	83%
4	10	1.5m	84%
4	100	1.6m	91%
5	10	1.9m	93%
5	512	1.8m	95%
6	1000	2.6m	97%
8	100 & 50	4.7m	12%
8	1000 & 500	5.7m	91%

Table 2: Model Configuration vs Accuracy

4 Build Basic CNNs

The goal of this phase was to build a series of basic CNNs (not using data augmentation, regularization, etc.) that performs best against validation data. Given I ran many models when trying to overfit, I brought a few of the highest accuracy models forward for testing on validation. I also noticed a parameter difference when using Flatten() versus GlobalAveragePooling2D() so I studied how these two layers work as well.

4.1 GlobalAverage vs Flatten

I experimented with this architecture by switching Flatten for GlobalAveragePooling to check the differences.

Layer (type)	Output Shape	Param #
conv2d_26 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_25 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_27 (Conv2D)	(None, 72, 72, 64)	18,496
max_pooling2d_26 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_28 (Conv2D)	(None, 34, 34, 128)	73,856
max_pooling2d_27 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_29 (Conv2D)	(None, 15, 15, 256)	295,168
max_pooling2d_28 (MaxPooling2D)	(None, 7, 7, 256)	0
conv2d_30 (Conv2D)	(None, 5, 5, 512)	1,180,160
max_pooling2d_29 (MaxPooling2D)	(None, 2, 2, 512)	0
global_average_pooling2d_4 (GlobalAveragePooling2D)	(None, 512)	0
dense_10 (Dense)	(None, 200)	102,600
dense_11 (Dense)	(None, 8)	1,608

Total params: 1,672,784 (6.38 MB)
Trainable params: 1,672,784 (6.38 MB)
Non-trainable params: 0 (0.00 B)

(a) GlobalAveragePooling

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_1 (Conv2D)	(None, 72, 72, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_2 (Conv2D)	(None, 34, 34, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_3 (Conv2D)	(None, 15, 15, 256)	295,168
max_pooling2d_3 (MaxPooling2D)	(None, 7, 7, 256)	0
conv2d_4 (Conv2D)	(None, 5, 5, 512)	1,180,160
max_pooling2d_4 (MaxPooling2D)	(None, 2, 2, 512)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 200)	409,800
dense_1 (Dense)	(None, 8)	1,608

Total params: 1,979,984 (7.55 MB)
Trainable params: 1,979,984 (7.55 MB)
Non-trainable params: 0 (0.00 B)

(b) Flatten

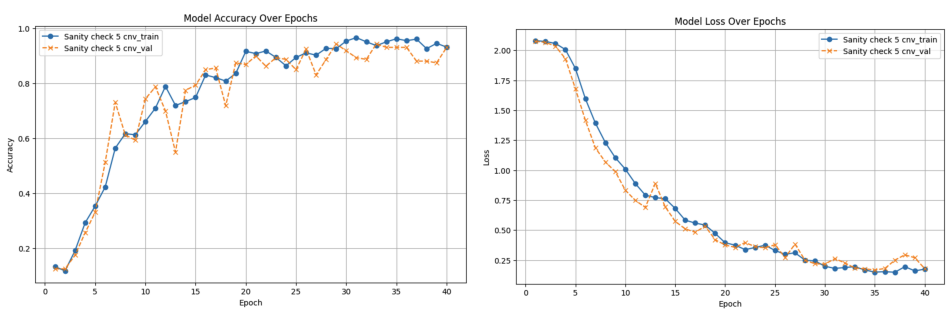
Figure 2: Parameter count of Flatten and GlobalAveragePooling

Take note of the parameters specifically in the Dense layer. Why did this happen? Flatten works by taking the 512 output depth feature map of 2 x 2 and flattening all of it to 1D tensor. So, to calculate, 2 x 2 x 512 = 2048 which is then passed into Dense. The calculation for that is 2048 x 200 (neurons in Dense) = 409,600 then add the bias for each dense neuron giving us the 409,800 parameters. GlobalAveragePooling has a slightly different approach where it calculates the average of each 2 x 2 feature map. Since this average is over 512 feature maps, this results in an output of (512,). To finish the calculation, I take 512 x 200 + 200 (each neuron's bias) = 102,600. However,

even with this difference in parameters, the accuracy was not greatly effected in this particular example. Keep in mind that larger feature maps can result in a greater difference in parameters which could then have an effect.

4.2 Model Comparisons

After running several models and checking the validation accuracy I took the best two and implemented callbacks. The monitor metric for both early-stopping and model checkpoint is validation loss. I also set the patience parameter to 5 epochs in early stopping because my results were spiking. I did not want the model to stop on one weird spike in validation loss because I felt the spikes were useful information for me to learn. I also implemented the early stop to restore the best weights because of this. Meaning, if the model continued to run on patience and deteriorated or had a spike, I would still have the best performing weights at the end of its best epoch. The two models I ran these callbacks on are the 5 and 6 convolutional layer models. The 6 layer had a test accuracy of 92.5% while the 5 layer tested at 97.2%. Below are the curves of the 5 layer model



(a) Accuracy over epochs (b) Loss over epochs

Figure 3: 5 conv layers - 200 dense - 1.6m parameters

The chart below shows the metrics on the test dataset per class. The metrics chosen are common in machine learning, precision, recall, and f-1 score. Precision means true positives. So out of all the bases classified as a certain town hall - how many of those were actually that town hall. Recall is based on the accuracy of correctly predicting all town halls of a certain class. Lets say there are 20 Town Hall 14, of those 20 how many did we correctly classify. F-1 is a formula of both recall and precision. Looking at the chart

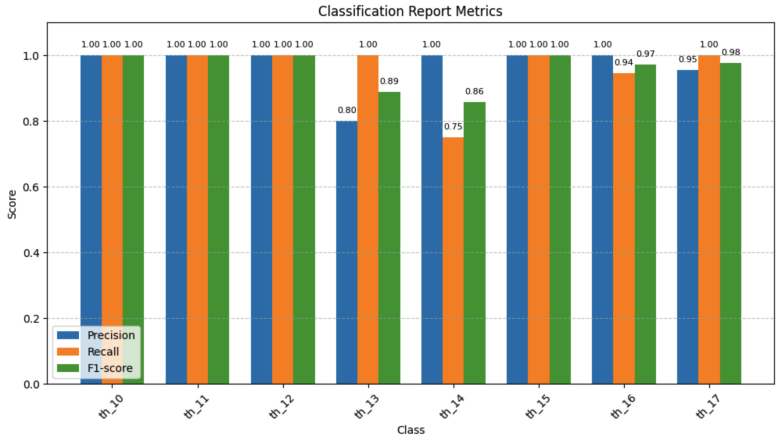


Figure 4: Per class metrics

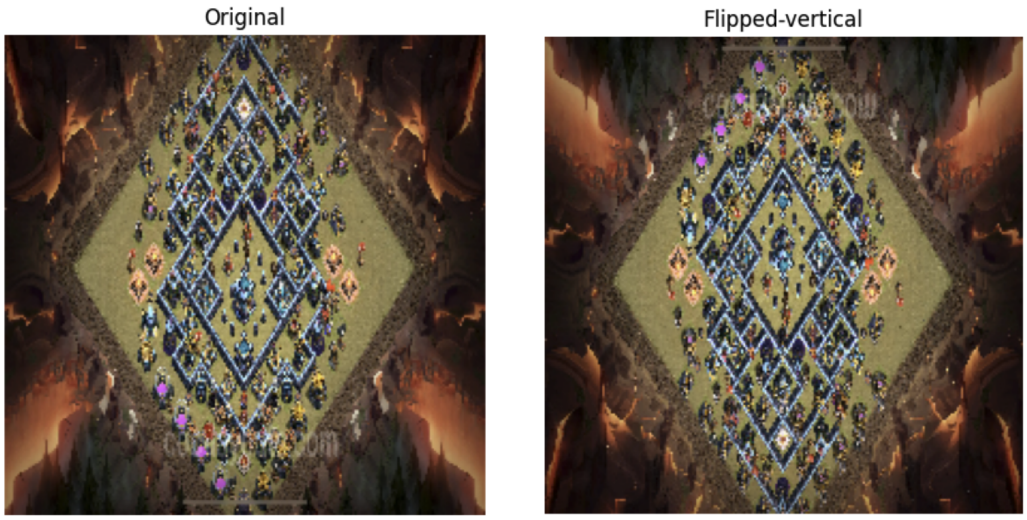
it appears like the model did very well for all but a few Town Hall levels. Given Town Hall 13 has a low precision, which means we guessed some Town Halls as Town Hall 13 and they were not actually Town Hall 13, and Town Hall 14 has a low recall, which means we missed correctly classifying them. It is likely the model misclassified some Town hall 14s as Town Hall 13. It also appears likely the model mistook some Town Hall 16s as Town Hall 17. Its easier to make this guess because the separation of scores. Town Hall 15 looks drastically different than 13 and 14 and scored perfect. This can also occur because of the how the game works. When a player upgrades a Town Hall the only building to change initially is the Town Hall itself. All other buildings would be of the previous Town Hall level and I intentionally put images that exhibit this in the dataset.

5 Augmentation

Augmentation applies different types of transformation to images such as rotating, flipping vertically or horizontally, zooming, and more. This helps create more training data for the model to learn from. While I expected a positive outcome from augmenting given my smaller (1042 images, 8 classes) data set, I experienced some overfitting when trying any augmenting except for vertical and horizontal flips. Augmentation had to be done carefully because the freedom of Town Hall placement. If a shift or zoom caused the

Town Hall to be out of scope that would be detrimental to training which could explain why all but vertical and horizontal flipping of images hurt the model's performance. It should be noted augmentation adds no parameters to the model.

5.1 Examples of Augmenting

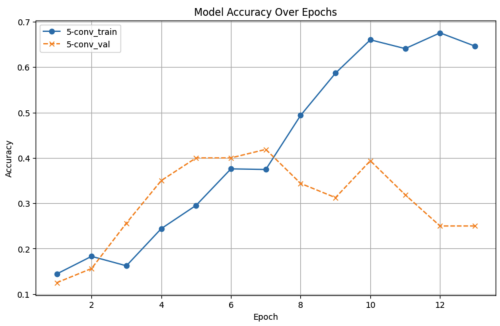


(a) Base with no augmentation (b) Vertical flip augmentation

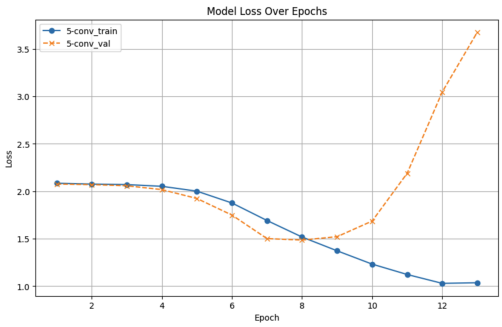
Figure 5: Augmentation examples

Displayed above is an example of vertical augmentation. Ironically, flipped bases do exist in Clash and can be common among average players (non-tournament players). Usually these mirror-image bases are more so flipped over a horizontal axis but at least augmenting can still represent a real-ish possibility.

5.2 Effects of Augmenting

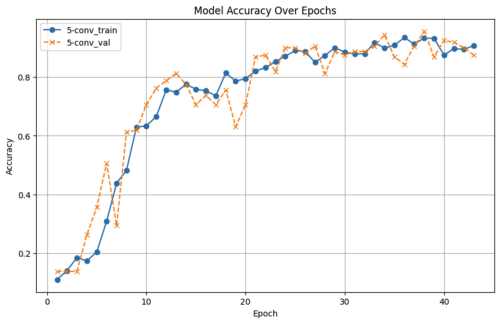


(a) Accuracy

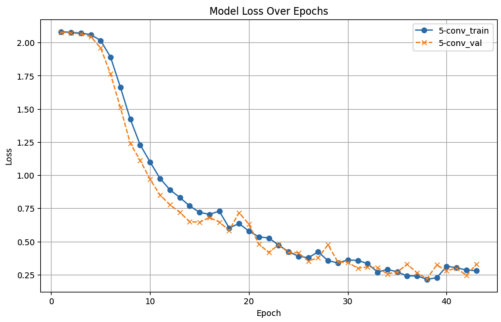


(b) Loss

Figure 6: Augmentation with height and width shifts and brightness changes



(a) Accuracy



(b) Loss

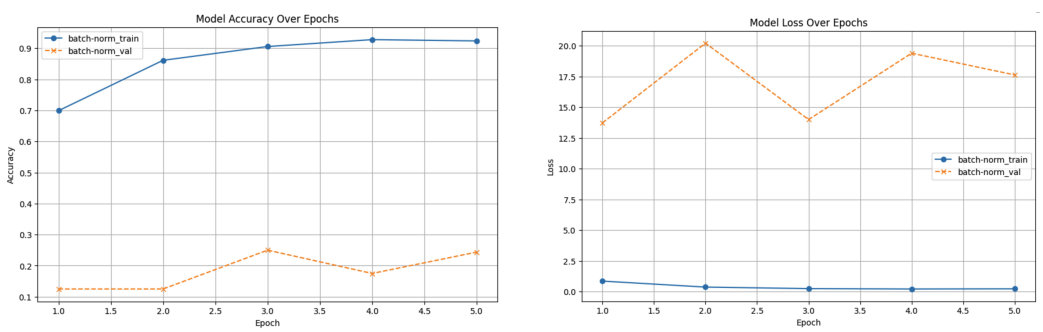
Figure 7: Augmentation only using vertical and horizontal flip

As seen above, too much augmentation poorly effects the model. This dataset appers sensitive to too much changes in the images. When only using the vertical and horizontal flipping parameters the model performs well but not as positive as the base model without augmentation.

6 Regularization

The goal of this phase is to test the effects of regularization techniques on the Clash of Clans data set. These include dropout, batch normalization, and regularizers. Normalization is a common technique in machine learning that tries to normalize the data over a range, usually done by subtracting the mean and dividing by the standard deviation. This centers the data around 0 and usually results in a fairly nice bell curve. It should be noted that Chollet says, "No one really knows for sure why batch normalization helps"[CW25]. It is meant to normalize the data as it changes throughout training. Dropout randomly drops out input to reduce noise. Regularization adds penalty depending on which method is used but for this task I used L2 on the kernels.

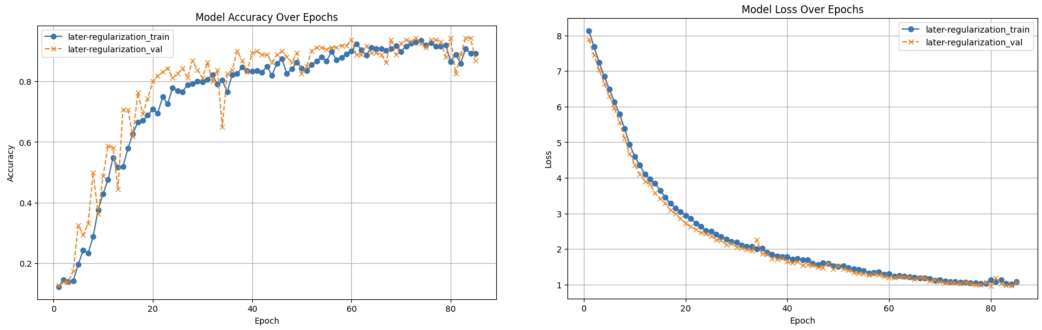
6.1 Effects of Regularization



(a) Accuracy (b) Loss

Figure 8: 1 Batch Normalization layer

The first attempt included adding a BatchNormalization layer after each convolution but that resulted in immediate overfitting so I tuned it down to one BatchNormalization at the end of the network. However, as seen in the charts, this did not fix the overfitting problem with using BatchNormalization on its own. Unlike augmentation, batch normalization adds additional parameters. However, these can be set in Keras to be trainable parameters or non-trainable parameters. In any case, they do not add a large amount of parameters.



(a) Accuracy (b) Loss

Figure 9: Regularized late layers

The best performance came from kernel regularization. Similar to Batch-Normalization, too much regularization also resulted in bad performance but also very slow convergence. Adding regularization at the end of the network only on the 512 filter conv layer with Dropout(0.2) did help and the loss curve is much smoother, but the model still suffers validation accuracy spikes. They smoothen out toward the end of training, but overall it did not beat the baseline model and in fact tied it exactly on validation data.

6.2 Table of Validation Accuracy

Regularization Type	**Accuracy / Result**
BatchNormalization all layers	Overfit
BatchNormalization last layer	Overfit
Regularized (L2, Dropout) all layers	75.63%
Regularized (L2, Dropout) last layers	94.38%
Normal model	94.38%

Table 3: Comparison of Regularization Strategies and Their Effect on Model Accuracy

7 Residual Networks

The goal of this phase is to test the effects of using residual and pretrained networks such as ResNet, VGG, etc. Residual networks add inputs into their outputs. The trick here is keeping track of the shape. Conv with increasing filters and MaxPooling change shapes so the implementation has to be adjusted slightly in those situations. Pretrained networks involve utilizing a model that has already been trained and then fine-tuning it to your data set. When doing this you can alter any layers parameters to be trainable or not.

7.1 Effects of Residual Connections

When using residual connections the graph of a model will look a bit different. Instead of being perfectly linear there are now some extra branches each time I utilize the "add" method from layers. A graph of residual connections looks like the image below.

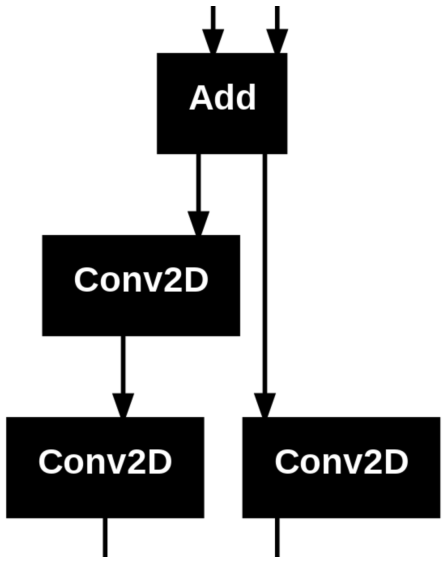
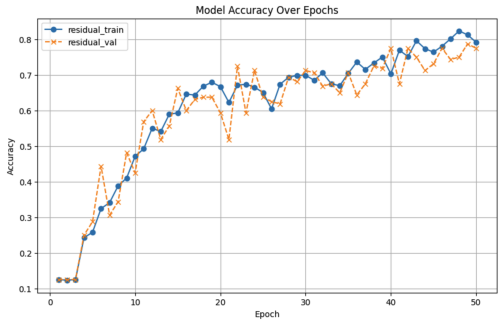
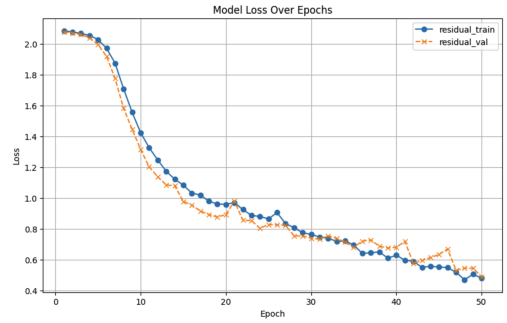


Figure 10: Residual Connections

Residual Networks performed better than regularization but still suffer the same spikes in training. Adding residual layers resulted in a best epoch validation accuracy of just under 80%.



(a) Accuracy

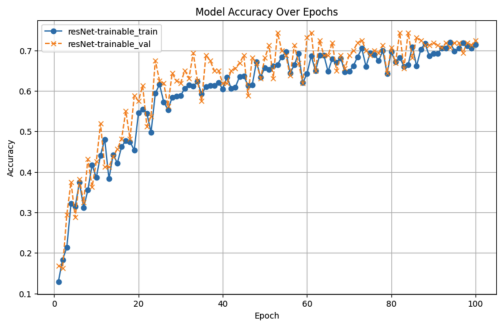


(b) Loss

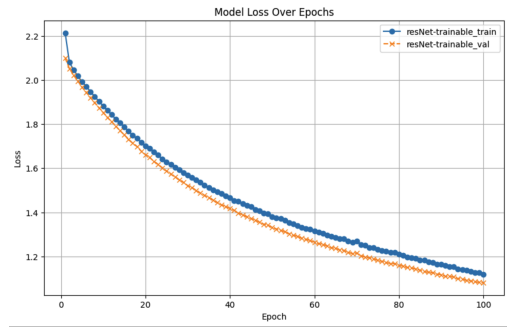
Figure 11: Residual connection metrics

7.2 ResNet

In this section I tried utilizing the ResNet50 from Keras. I ran 3 different tests - one in which no layers were trainable and then two with different amounts of ResNet layers being trainable. I noticed very quickly the parameters will skyrocket if even a couple layers are made trainable and the model overfits quickly. The loss curve is incredibly smooth but the validation spikes seem to be more frequent than previous runs. They do taper out toward the end of the model but overall its still not great. The model also converges very slowly. The final model tested ran for 100 epochs and did not reach early stopping at that time while achieving only 73% validation accuracy.



(a) Accuracy



(b) Loss

Figure 12: ResNet metrics

8 Conclusion

In summary, I learned of the importance for each process of building a convolutional neural network. I identified the strengths and weaknesses of various techniques relative to my dataset. Trying to overfit at first became very useful in later tasks as the models which scored higher than 90% ended up performing well when being run against validation data. I built my baseline model, tested it against test, examined the metrics per class, and then used validation accuracy to compare against other techniques. I showed my dataset was sensitive to various types of more aggressive augmentation, while simple augmentation performed the best. Applying BatchNormalization had to be done conservatively to avoid overfitting and adding regularization with Dropout to the later layers did give the best results of all the techniques except the baseline simple-ish model. Overall, I would choose to stick with the 5 convolution - 200 dense - 1.6m parameter model when compared with the regularized model because the time to train when adding regularization increased quite a bit. The best impacts actually came from simply decreasing the learning rate to 0.0001 and keeping the padding parameter set to "same". I learned building a model seems to be more of an art than science. Datasets are unique and what works for one doesn't work for all.

References

[CW25] François Chollet and Matthew Watson. *Deep Learning with Python*. Third. Version 3, last updated: 2025-04-07. Manning Publications, 2025.